Proc.9th Int. Conf. on Information and Knowledge Management CIKM 2000, Washington, DC.

High Performance Clustering Based on the Similarity Join

Christian Böhm, Bernhard Braunmüller, Markus Breunig, Hans-Peter Kriegel
Institute for Computer Science
University of Munich
Oettingenstr. 67, 80538 München, Germany
Phone +49-89-2178-2190
{boehm,braunmue,breunig,kriegel}@dbs.informatik.uni-muenchen.de

Abstract

A broad class of algorithms for knowledge discovery in databases (KDD) relies heavily on similarity queries, i.e. range queries or nearest neighbor queries, in multidimensional feature spaces. Many KDD algorithms perform a similarity query for each point stored in the database. This approach causes serious performance degenerations if the considered data set does not fit into main memory. Usual cache strategies such as LRU fail because the locality of KDD algorithms is typically not high enough. In this paper, we propose to replace repeated similarity queries by the similarity join, a database primitive prevalent in multimedia database systems. We present a schema to transform query intensive KDD algorithms into a representation using the similarity join as a basic operation without affecting the correctness of the result of the considered algorithm. In order to perform a comprehensive experimental evaluation of our approach, we apply the proposed transformation to the clustering algorithm DBSCAN and to the hierarchical cluster structure analysis method OPTICS. Our technique allows the application of any similarity join algorithm, which may be based on index structures or not. In our experiments, we use a similarity join algorithm based on a variant of the X-tree. The experiments yield substantial performance improvements of our technique over the original algorithms. The traditional techniques are outperformed by factors of up to 33 for the X-tree and 54 for the R*-tree.

Keywords

Data mining, clustering, database primitives, similarity join, multidimensional index structure.

1 Motivation

Knowledge discovery in databases (KDD) is the complex process of extracting implicit, previously unknown and potentially useful information from data in databases [41]. In recent years, KDD has gained increasing interest in the research community as well as in traditional business data processing. The field of KDD knows various standard tasks such as classification [37], mining association rules [3], trend detection [11], and visualization [24]. One of the most important tasks among those is *clustering* [23]. The goal of a clustering algorithm is to group the objects of a database into a set of meaningful subclasses (clusters), such that objects in the same cluster are more similar to each other than objects belonging to different clusters. There are numerous applications of clustering [42, 39, 26, 12].

Recently, algorithms for extracting knowledge from large *multidimensional* data sets have become more and more important due to the fact that multidimensional data are prevalent in numerous non-standard applications for database systems. Examples of such systems include CAD databases [22], medical imaging [32] and molecular biology [31].

When considering algorithms for KDD, we can observe that many algorithms rely heavily on repeated *similarity queries*, i.e. range queries or nearest neighbor queries, among feature vectors. For example, the algorithm for mining spatial association rules proposed in [25] performs a similarity query for each object of a specified type, such as a town. For various other KDD algorithms, this situation comes to an extreme: a similarity query has to be answered *for each object in the database* which obviously leads to a considerable computational effort.

In order to accelerate this massive similarity query load, multidimensional index structures [10, 33, 4] are usually applied for the management of the feature vectors. Provided that the index quality is high enough, which can usually be assumed for low and medium dimensional data spaces, such index structures accelerate the similarity queries to a logarithmic complexity. Therefore, the overall runtime complexity of the KDD algorithm is in $O(n \log n)$. Unfortunately, the overhead of executing all similarity queries separately is large. The locality of the queries is often not high enough, so that usual caching strategies for index pages such as LRU fail, which results in serious performance degenerations of the underlying KDD algorithms. Several solutions to alleviate this problem have been proposed, e.g. sampling [16] or dimensionality reduction [14]. These techniques imply some loss of information which may not be acceptable in some application domains. Both can, however, also be applied to our approach as a preprocessing step.

The basic intention of our solution is to substitute the great multitude of expensive similarity queries by a *similarity join operation* using a distance-based join predicate, without affecting the correctness of the result of the given KDD algorithm: Consider a KDD algorithm that performs a range query (with range ε) in a large database of points P_i ($0<i<n$) for a large set of query points Q_j ($0<j<m$). During the processing of such an algorithm, each point P_i in the database is combined with each query point Q_j which has a distance of no more than ε. This is essentially a join operation between the two point sets P and Q with a distance-based join predicate, a so-called *distance join* or *similarity join*. The general idea of our approach is to transform query intensive KDD algorithms such that the transformed algorithms are based on a similarity join instead of repeated similarity queries. In this paper, we

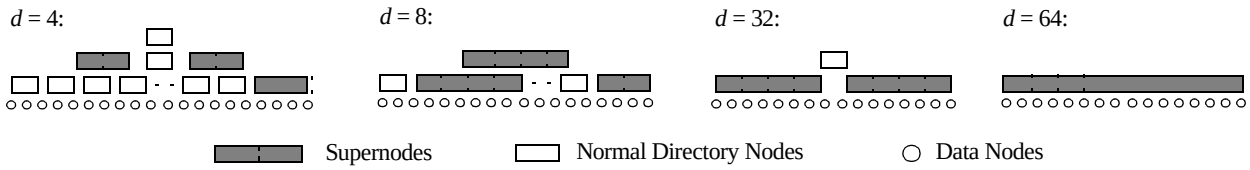

Figure 1. : Structure of the X-tree with respect to the dimension

concentrate on algorithms which perform a range query for each point in the database. In this case, the similarity join is a self-join on the set of points stored in the database. Nevertheless, our approach is also applicable for many other KDD algorithms where similarity queries are not issued for each database object, but which are still query intensive. Additionally, since a large variety of efficient processing strategies have been proposed for the similarity join operation, we believe that our approach opens a strong potential for performance improvements.

Note that this idea is not applicable to every KDD algorithm. There is a class of algorithms which is not meant to interact with a database management system and thus is not based on database primitives like similarity queries, but instead works directly on the feature vectors. What we have in mind is the large class of algorithms which are based on repeated similarity queries (or, at least, can be based on similarity queries). Examples of methods where our idea can be applied successfully are the distance based outlier detection algorithm RT [27], the density based outliers LOF [7], the clustering algorithms DBSCAN [13], DenClue [19], OPTICS [1], nearest-neighbor clustering [20], single-link clustering [46], spatial association rules [25], proximity analysis [26], and other algorithms. In this paper, we demonstrate our idea on the known clustering algorithm DBSCAN and on the recently proposed hierarchical clustering method OPTICS.

The remainder of our paper is organized as follows: Section 2 describes related work. Section 3 proposes a schema for transforming KDD algorithms using repeated range queries into equivalent algorithms using similarity joins. In section 4, we present a comprehensive experimental evaluation of our technique, and section 5 concludes our paper.

2 Related Work

2.1 Clustering Algorithms

Existing clustering algorithms can be classified into *hierarchical* and *partitioning clustering* algorithms (see e.g. [23]). Hierarchical algorithms decompose a database *D* of *n* objects into several levels of nested partitionings (clusterings). Partitioning algorithms, on the other hand, construct a flat (single level) partition of a database *D* of *n* objects into a set of *k* clusters such that the objects in a cluster are more similar to each other than to objects in different clusters. Popular hierarchical algorithms are e.g. the *Single-Link method* [46] and its variants (see e.g. [23, 38]) or CURE [16]. Partitioning methods include *k-means* [36], *k-modes* [21], *k-medoid* [28] algorithms and *CLARANS* [39]. The basic idea of partitioning methods is to determine the set of pairwise distances among the points in the data set. Points with minimum distances are successively combined into clusters.

Density based approaches apply a local cluster criterion and are popular for the purpose of data mining, because they yield very good quality clustering results. Clusters are regarded as regions in the data space in which the objects are dense, separated by regions of low object density. The local densities are determined by *repeated range queries*. We can distinguish between algorithms that execute these range queries directly and algorithms that replace these range queries by a grid approximation.

Repeated range queries are executed directly in the DBSCAN algorithm [13]. While DBSCAN as a partitioning algorithm computes only clusters of one given density, *OPTICS* [1] generates a density based cluster-ordering, representing the intrinsic hierarchical cluster structure of the data set in a comprehensible form. Both algorithms execute exactly one ε-range query for every point in the database.

Due to performance considerations several proposals rely on grid cells [23] to accelerate query processing, e.g. *WaveCluster* [45], *DenClue* [19] and *CLIQUE* [2]. Some of them, e.g. DenClue, can be easily transformed into an equivalent form executing multiple similarity queries instead, and, thus, our method can be applied.

2.2 Indexing Multidimensional Spaces

A wide variety of spatial access methods (SAM) have been proposed in the literature. In [15] an extensive overview can be found. Among these SAMs is the R-tree [17] with its variants, e.g. the R+-tree [44], the R*-tree [10] and the X-tree [4]. In this subsection, we will briefly review the R*-tree and the X-tree, since these will be the SAMs that we use for our experimental evaluation.

The R-tree is an extension of the B^+-tree for multidimensional data objects. Leaf nodes of the R-tree contain entries of the form (*oid*, *MBB*) where *oid* is an object identifier pointing to the exact object representation and *MBB* is the *n*-dimensional minimal bounding box enclosing the corresponding data object. Non-leaf nodes contain entries of the form (*child*, *MBB*) where *child* is a pointer to a successor node in the next lower level of the R-tree and *MBB* is a minimal bounding box which covers all entries in the child node. The R-tree uses the concept of overlapping regions, i.e. nodes on the same level are allowed to overlap. From the variants of the original R-tree, the R*-tree seems to offer the best query performance for moderate dimensions. The main idea of the R*-tree is to use *forced reinserts* which defers splits by first reinserting some data objects lying in the overflowing node. Additionally, an improved node splitting policy which also considers overlap and region perimeters leads to a better structure of the directory. Based on a detailed study of the R*-tree behavior when dealing with high-dimensional data objects, Berchtold et al. proposed the X-tree (eXtended node tree). The most important aspect of the X-tree is the concept of directory supernodes. Whenever the split of a directory node would lead to a high overlap of the resulting nodes or to overlap minimal but extremely unbalanced nodes, the overflowing node is transformed into a supernode, i.e. a node with a larger than usual block size. A supernode is split (and possibly retransformed into two ordinary directory nodes) whenever an overlap minimal and balanced split exists. Another important property of the X-tree is an advanced split algorithm, which provides (at least for point data) an overlap free split using the history of previous splits. In figure 1 the structure of the X-tree is depicted for various

dimensions. We can observe that with increasing dimension the directory of the X-tree degenerates to few and eventually to one large supernode containing the lowest directory level of the corresponding *R*-tree.

2.3 Similarity Join Algorithms

A join combines the tuples of two relations *R* and *S* into pairs such that a *join predicate* is fulfilled. In multidimensional databases, *R* and *S* contain points (feature vectors) rather than ordinary tuples. In a *similarity join*, the join predicate is similarity, i.e. the distance between two feature vectors stored in *R* and *S* must not exceed a threshold value ε in order to appear in the result set of the join. If *R* and *S* are actually the same relation, the join is called a *self-join*.

Basic Strategies

The simplest way to evaluate a given join is the tuplewise nested loop strategy [48]. One loop iterates over every tuple of *R* and a second nested loop iterates over every point in *S*. Thus, the join predicate is evaluated for every pair of points. This strategy can be improved by changing the order in which the points are combined. For this purpose, a part of the database cache is reserved for the points in *R*. Instead of reading *R* tuple by tuple in the outer loop, it is read in blocks fitting in the reserved part of the cache. For each block in *R*, the relation *S* is scanned. This strategy reduces the number of times the relation *S* is scanned. If the join predicate can be supported by an appropriate index, the nested loop join can be improved by replacing the inner loop by an indexed access to all points of *S* matching (with respect to the join predicate) a given point of *R*. This evaluation strategy is called *indexed nested loop join*.

Advanced Join Algorithms Using R-trees

Most related work on join processing using multidimensional index structures is based on the *spatial join*. The spatial join operation is defined for 2-dimensional polygon databases where the join predicate typically is the intersection between two objects. This kind of join predicate is prevalent in map overlay applications. We adapt the algorithms to allow distance based predicates for point databases instead of the intersection of polygons.

The most common technique is the *R-tree Spatial Join* (RSJ) [8], which is based on R-tree like index structures constructed on *R* and *S*. RSJ is based on the lower bounding property, which states that the distance between two points is never smaller than the distance between the regions of the two pages in which the points are stored. The RSJ algorithm traverses the indexes for *R* and *S* synchronously. When a pair of directory pages (P_R,P_S) is under consideration, the algorithm forms all pairs of the child pages of P_R and P_S having distances of at most ε. For these pairs of child pages, the algorithm is called recursively. Thus, the corresponding indexes are traversed in a depth-first order.

Various optimizations of RSJ have been proposed. Huan, Jing and Rundensteiner propose the BFRJ-algorithm [18] which traverses the indexes according to a breadth-first strategy. In [9] the authors adapt RSJ for join processing on parallel computers using shared virtual memory. Their technique improves both CPU and I/O-time.

Other Advanced Join Algorithms

If no multidimensional index is available, it is possible to construct the index on the fly before starting the join algorithm. Usually, the dynamic index construction by repeated insert operations performs poorly and cannot be amortized by performance gains during join processing. Several techniques for bulk-loading multidimensional index structures have been proposed [5, 6]. Their runtime is substantially lower than the runtime of the repeated insert operations, even if the data set does not fit into main memory. This effort is typically amortized by efficiency gains in the join.

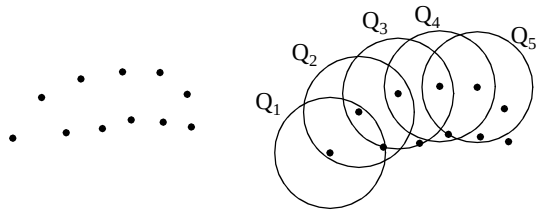

Figure 2. Sequence of Range Queries for A_1

The seeded tree method [34] joins two point sets provided that only one is supported by an R-tree. The partitioning of this R-tree is used for a fast construction of the second index on the fly. This approach is not applicable for self-joins. The spatial hash-join [35, 40] divides the set *R* into a number of partitions according to system parameters. Sampling is applied to determine initial buckets. If each bucket fits in main memory, a single scan of the buckets is sufficient to determine all join pairs. The join based on the ε-KDB-tree [47] uses tree matching. One ε-KDB-tree is built on the fly for each relation and the given ε. The ε-KDB-tree is a variant of the KDB-tree [43]. The Size Separation Spatial Join and the Multidimensional Spatial Join are presented in [29, 30]. These approaches consider each point as a cube with side-length ε and make use of space filling curves to order the points in a multidimensional space.

3 Similarity-Join Based Clustering

In section 2, we have seen that density based clustering algorithms perform range queries in a multidimensional vector space. Since a range query is executed for each point stored in the database, we can describe those algorithms using the following schema A_1:

Algorithmic Schema A_1:

```
foreach Point p ∈ D {
    PointSet S = RangeQuery (p, ε) ;
    foreach Point q ∈ S
        DoSomething (p,q) ;
}
```

In order to illustrate this algorithmic schema, we consider as an example task the determination of the core point property for each point of the database. According to the DBSCAN definition, a point is a core point if there is at least a number of *MinPts* points in its ε-neighborhood (for a formal definition see [13]). For this task, the procedure DoSomething (*p,q*) will simply increment a counter and set the core point flag if the threshold *MinPts* is reached. Assume a sample data set with one cluster as depicted on the left side of figure 2. On the right side of figure 2 is the start of a sequence order in which schema A_1 may evaluate the range queries. Since A_1 does not use the information which points belong to which page of the index, the sequence of the range queries does not consider the number of page accesses or even optimize for a low number of page accesses.

Under the assumption of a page capacity of 4 data points, a pagination as depicted in figure 3 is quite typical and, for our sample sequence, the following page accesses must be performed: Query Q_1 accesses page P_1 and the queries Q_2 and Q_3 both access the pages P_1 and P_2. The query Q_4 accesses all three pages P_1, P_2 and P_3, and so on. After processing the upper part of the cluster, range queries for the lower part are evaluated and thus P_1 is accessed once again. But at this point in time, P_1 is eventually discarded from the cache and therefore P_1 must be loaded into main memory again.

However, by considering the assignment of the points to the pages, a more efficient sequence for the range queries can be derived, i.e. loading identical data pages several times into main memory can be avoided: First, determine all pairs of points on page P_1 having a distance no more than ε; then, all pairwise distances of points on page P_2; and afterwards, all cross-distances between points on page P_1 and P_2. Then, P_1 is no longer needed and can be deleted from the cache. Finally, we load page P_3 from secondary storage and determine the pairs on P_3 and the cross-distances between P_2 and P_3. Since the distance between the pages P_1 and P_3 is larger than ε, there is no need to determine the corresponding cross-distances. Processing the data pages in this way clearly changes the order in which data points with a distance no more than ε are combined. The only difference from an application point of view, however, is that we now count the ε-neighborhoods of many points *simultaneously*. Therefore, we simply need an additional attribute for each point which may be a database attribute unless all active counters fit into main memory.

What we have actually done in our example is to transform the algorithmic schema A_1 into a new algorithmic schema A_2 and to replace the procedure DoSomething (*p*,*q*) by a new, but quite similar procedure DoSomething' (*p*,*q*). The only difference between these two procedures is that the counter which is incremented in each call is not a global variable but an attribute of the tuple *p*. The changes in the algorithmic schema A_2 are more complex and can be expressed as follows:

Algorithmic Schema A_2:

```
foreach DataPage P {
    LoadAndFixPage (P) ;
    foreach DataPage Q
        if (mindist (P,Q) ≤ ε) {
            CachedAccess (Q) ;
            /*  Run Algorithmic Schema A1 with    */
            /*  restriction to the points on P and Q:  */
            foreach Point p ∈ P
                foreach Point q ∈ Q
                    if (distance (p,q) ≤ ε)
                        DoSomething' (p,q) ;
        }
    UnFixPage (P) ;
}
```

Here, mindist (P,Q) is the minimum distance between the page regions of P and Q, i.e.

$$\text{mindist}^2(P, Q) = \sum_{0 \le i < d} \begin{cases} (P.\text{lb}_i - Q.\text{ub}_i)^2 & \text{if } P.\text{lb}_i > Q.\text{ub}_i \\ (Q.\text{lb}_i - P.\text{ub}_i)^2 & \text{if } Q.\text{lb}_i > P.\text{ub}_i \\ 0 & \text{otherwise} \end{cases}$$

where lb_i and ub_i denote the lower and upper boundaries of the page regions. CachedAccess (...) denotes the access of a page through the cache. Thus, a physical page access is encountered if the page is not available in the cache. In order to show the correctness of this schema transformation, we prove the equivalence of schema A_1 and A_2 in the following lemma.

Lemma: Equivalence of A_1 and A_2.

(1) The function DoSomething' is called for each pair (*p*,*q*) in the algorithmic schema A_2 for which DoSomething is called in schema A_1.

(2) DoSomething is called for each pair (*p*,*q*) for which DoSomething' is called.

Proof:

(1) If DoSomething (*p*,*q*) is called in A_1, then *q* is in the ε-neighborhood of *p*, i.e. the distance $|p - q| \le \varepsilon$. The points are either stored on the same page *P* (case **a**) or on two different pages *P* and *Q* (case **b**).

(a) As mindist (*P*,*P*) = 0 ≤ ε the pair of pages (*P*,*P*) is considered in A_2. The pair of points (*p*,*q*) is then encountered in the inner loop of A_2 and, thus, DoSomething' (*p*,*q*) is called.

(b) As the regions of the pages *P* and *Q* are conservative approximations of the points *p* and *q*, the distance between the page regions cannot exceed the distance of the points, i.e. $\text{mindist}(P,Q) \le |p - q| \le \varepsilon$. Therefore, the pair of pages (*P*,*Q*) is considered in A_2 and DoSomething'(*p*,*q*) is called.

(2) If DoSomething' is called in A_2, then $|p - q| \le \varepsilon$. As *q* is in the ε-neighborhood of *p*, DoSomething (*p*,*q*) is called in A_1. **q.e.d.**

We note without a formal proof that for each pair (*p*,*q*) both DoSomething and DoSomething' are evaluated at most once. Considering the algorithmic schema A_2, we observe that this schema actually represents an implementation of a join-operation which is called *pagewise nested loop join*. More precisely, it is a self-join operation where the join predicate is the distance comparison $|p - q| \le \varepsilon$. Such a join is also called *similarity self-join*. If we hide the actual implementation (i.e. the access strategy of the pages) of the join operation, we could also replace the algorithmic schema A_2 by a more general schema A_3 where $D \lozenge\!\lozenge_{|p - q| \le \varepsilon} D$ denotes the similarity self-join:

Algorithmic Schema A_3:

```
foreach PointPair (p,q) ∈ (D ◊◊|p - q| ≤ ε D)
    DoSomething' (p,q) ;
```

This representation allows us not only to use the pagewise nested loop join but any known evaluation strategy for similarity joins (cf. section 2). Depending on the existence of an index or other preconditions, we can select the most suitable join implementation.

When transforming a KDD algorithm, we proceed in the following way: First, the considered KDD method is broken up into several subtasks that represent independent runs of the similarity join algorithm.

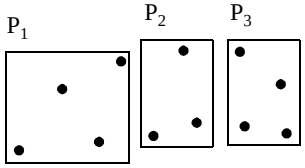

Figure 3. An Index Pagination for the Sample Data Set

Additional steps for preprocessing (e.g. index generation) and postprocessing (e.g. cleaning-up phases) may be defined. Then, the original algorithm in A_1 notation is transformed such that it operates on a cursor iterating over a similarity join (A_3 notation). Next, we consider how the operations can be further improved by exploiting the knowledge of not only one pair of points but of all points on a pair of index pages. In essence, this means that the original algorithm runs restricted to a pair of data pages.

In summary, our transformation of a KDD algorithm allows us to apply any algorithm for the similarity self-join, be it based on the sequential scan or on an arbitrary index structure. The choice of the join algorithm and the index structure is directed by performance considerations.

4 Experimental Evaluation

In order to show the practical relevance of our method, we applied the proposed schema transformation to two effective data mining techniques. In particular, we transformed the known clustering algorithm DBSCAN and the hierarchical cluster structure analysis method OPTICS such that both techniques use a similarity self-join instead of repeated range queries. Note again that the resulting cluster structures generated by DBSCAN and OPTICS based on the similarity self-join are identical to the cluster structures received from the original techniques. We performed an extensive experimental evaluation using two real data sets: an image database containing 64-*d* color histograms of 112,000 TV-snapshots and 300,000 feature vectors in 9-*d* representing weather data, both with the Euclidean distance. We used the original version of the R*-tree and a 2-level variant of the X-tree. In all experiments, the R*-tree and the X-tree were allowed to use the same amount of cache (10% of the database size). Additionally, we implemented the similarity query evaluation based on the sequential scan. The join algorithm we used is similar to the algorithm proposed in [8], i.e. the basic join strategy for R-tree like index structures. Advanced similarity join algorithms can further improve the performance of our approach. All experiments were performed on an HP-C160 under HP-UX B.10.20. In the following, **Q**-DBSCAN denotes the original algorithm, i.e. when DBSCAN is performed with iterative range queries, and **J**-DBSCAN denotes our new approach, i.e. based on a similarity self-join. In the same way we will use **Q**-OPTICS and **J**-OPTICS. In all experiments, we report the total time (i.e. I/O plus CPU time). The sequential scan methods on the file were implemented efficiently, such that the file is scanned in very large blocks. Therefore, the I/O cost of scanning a file is considerably smaller than reading the same amount of data pagewise from an index.

4.1 Page Size

In our first set of experiments, we performed DBSCAN and OPTICS with varying page sizes in order to determine the optimal page sizes with respect to the used access method. In figure 4a, the runtimes of **Q**-DBSCAN and **J**-DBSCAN on 100,000 points from the weather data with $\varepsilon = 0.005$ and *MinPts* = 10 are shown. The page size is given as the average number of points located on a data page. We can observe that for all page sizes the runtime of **Q**-DBSCAN is considerably higher than the runtime of **J**-DBSCAN and this holds for the R*-tree, for the X-tree and for the sequential scan. The speed-up factor of **J**-DBSCAN compared to **Q**-DBSCAN for the optimal page sizes is 20 for both index structures, i.e. **J**-DBSCAN based on the R*-tree is 20 times faster than **Q**-DBSCAN based on the R*-tree (and the same speed-up factor is achieved for the X-tree).

Performing **Q**-DBSCAN on the sequential scan clearly yields the worst runtime, which is 556 times the runtime of **J**-DBSCAN using the X-tree. Note that we used a logarithmic scale of the y-axis in figure 4 since otherwise the runtimes of **J**-DBSCAN and **J**-OPTICS would hardly be visible. Figure 4b shows the results for the image data. We clustered 40,000 points with $\varepsilon = 0.08$ and *MinPts* = 10. For this data set, the performance improvement of **J**-DBSCAN compared to **Q**-DBSCAN using the R*-tree is even higher: the speed-up factor is 54 when the R*-tree is the underlying access method and 19 using the X-tree. For small page sizes, performing **Q**-DBSCAN on the sequential scan yields a better runtime than using the R*-tree. However, when the page size of the R*-tree is well adjusted, the **Q**-DBSCAN on the sequential scan again has the slowest runtime. We can also observe, that the **J**-DBSCAN variants on the R*-tree and on the X-tree are relatively insensitive to page size

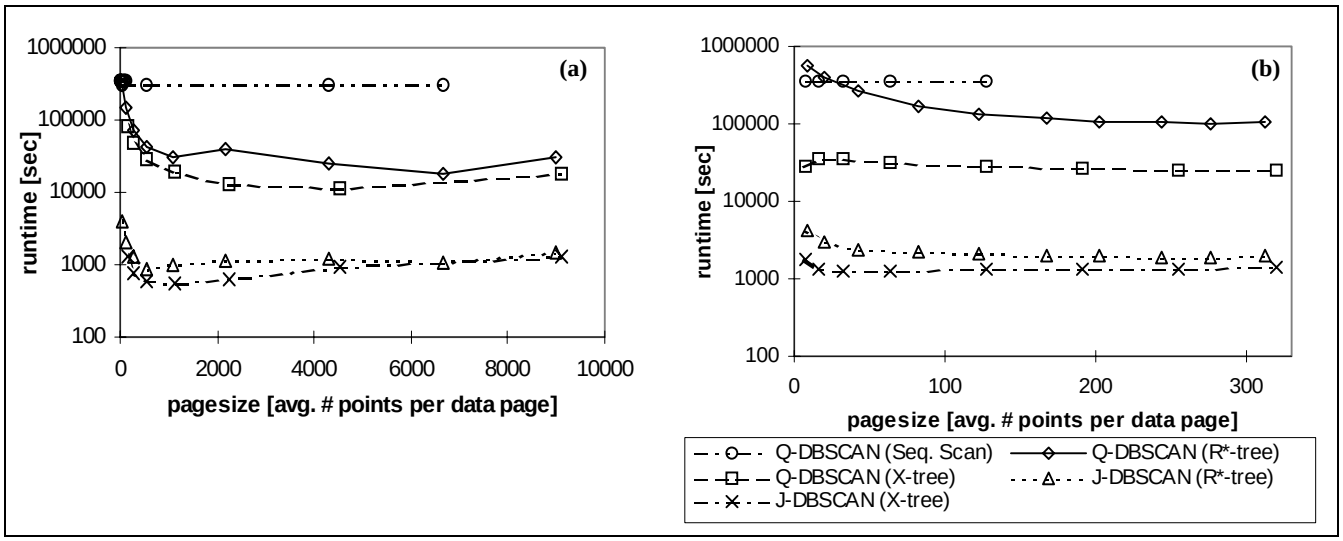

Figure 4. DBSCAN on (a) weather data (b) image data

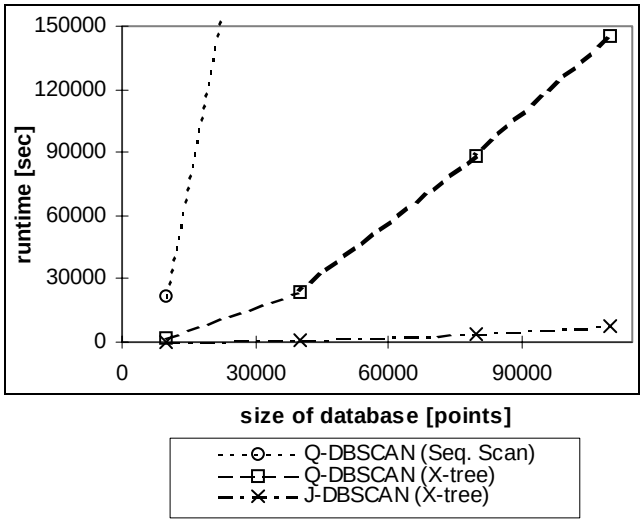

Figure 5. Scalability of DBSCAN on image data

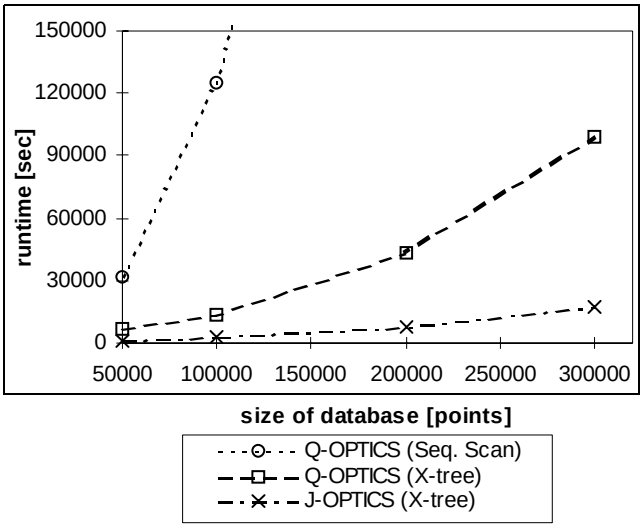

Figure 6. Scalability of OPTICS on weather data

calibrations. Independently from the underlying page size, the join based techniques outperform the query based techniques by large factors. Therefore, page size optimization is neither absolutely necessary to achieve good performance, nor is it possible to outperform our new techniques simply by optimizing the page size parameter of the query-based DBSCAN algorithm. For OPTICS, we observed similar results (not depicted). We varied the page size and ran **Q**-OPTICS and **J**-OPTICS on both data sets. The speed-up factor of **J**-OPTICS over **Q**-OPTICS is 22 using the R*-tree, and 12 using the X-tree.

Since the X-tree consistently outperformed the R*-tree on both data sets, we focus on the X-tree and on the sequential scan in the remainder of our experimental evaluation.

4.2 Database Size

Our next objective was to investigate the scalability of our approach when the database size increases. We ran **Q**-DBSCAN and **J**-DBSCAN with $\varepsilon = 0.005$, *MinPts* = 10 on the image data and increased the number of points from 10,000 to 110,000. As figure 5 depicts, the query based approach **Q**-DBSCAN scales poorly when the iterative range queries are processed by the sequential scan. The reason is that DBSCAN yields a quadratic time complexity when using a sequential scan as the underlying access method.

The scalability of **Q**-DBSCAN on top of the X-tree is obviously better due to the indexing properties of the X-tree. For **J**-DBSCAN, however, we clearly observe the best scalability as the database size increases: the speed-up factor compared to **Q**-DBSCAN using the X-tree increases to 20 for 110,000 points. The results on the weather data (not depicted) are very similar and the speed-up factor using the X-tree reaches 22 for 300,000 points.

We also investigated the scalability of **Q**-OPTICS and **J**-OPTICS. The results for the weather data with $\varepsilon = 0.01$ and *MinPts* = 10 are depicted in figure 6. In this experiment we increased the number of points from 50,000 up to 300,000. As before, the scalability of the query based approach is poor whereas the join based approach scales well. We also used the image data (not shown), increased the number of points from 10,000 to 110,000 and again found the scalability of **J**-OPTICS clearly better than the scalability of **Q**-OPTICS. For 110,000 points, the speed-up factor increases from 7.6 to 14 using the X-tree.

4.3 Query Range

For the performance of **Q**-DBSCAN and **Q**-OPTICS, the query range ε is a critical parameter when the underlying access method is an index structure. When ε becomes too large, a range query cannot be performed in logarithmic time since almost every data page has to be accessed. Consequently, performing **Q**-DBSCAN and **Q**-OPTICS on the sequential scan can yield better runtimes for large ε-values since the sequential scan does not cause random seeks on the secondary storage. In order to analyze our join based approach when ε becomes large, we ran **J**-DBSCAN and **J**-OPTICS with increasing ε-values.

Figure 7 depicts the results for **Q**-DBSCAN and **J**-DBSCAN on 40,000 points of the image data and *MinPts* = 10. The runtime of **Q**-DBSCAN drastically increases with ε whereas the runtime of **J**-DBSCAN shows only moderate growth, thus leading to a speed-up factor of 33 for $\varepsilon = 0.2$. Note, in figure 7 we omitted the runtimes of **Q**-DBSCAN on

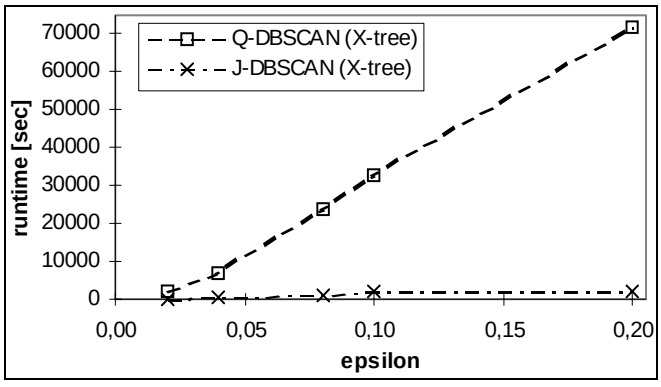

Figure 7. DBSCAN for increasing query range

the sequential scan since even for $\varepsilon = 0.2$ it was about 5 times the runtime of **Q**-DBSCAN based on the X-tree.

We also performed **Q**-OPTICS and **J**-OPTICS with increasing ε on 40,000 points of the image data and *MinPts* = 10 (cf. figure 8).The runtime of **Q**-OPTICS based on the X-tree quickly increases and eventually reaches the runtime of **Q**-OPTICS based on the sequential scan. Obviously, when increasing ε further, **Q**-OPTICS on the sequential scan will outperform **Q**-OPTICS based on the X-tree since the X-tree will read too many data pages for each range query while paying expensive random disk seeks. The runtime of **J**-OPTICS, on the other hand, shows a comparatively small growth when increasing ε. **J**-OPTICS outperforms **Q**-OPTICS on both the X-tree and the sequential scan by a large factor. This results from the fact that even when the similarity self-join generates all possible data page pairs due to a large ε, these are generated only once whereas **Q**-OPTICS generates these page pairs many times.

5 Conclusions

In this paper, we have presented a general technique for accelerating query-driven algorithms for knowledge discovery in databases. A large class of KDD algorithms depends heavily on repeated range-queries in multidimensional data spaces, which, in the most extreme case, are evaluated for every point in the database. These range queries are expensive database operations which cause serious performance problems when the data set does not fit into main memory. Our solution is the transformation of such a data mining technique into an equivalent form based on a similarity join algorithm. We proposed a general schema for rewriting KDD algorithms which use repeated range queries such that they are based on a similarity join. In order to show the practical relevance of our approach, we applied this transformation schema to the known clustering algorithm DBSCAN and to the hierarchical cluster structure analysis method OPTICS. The result of the transformed techniques are guaranteed to be identical to the result of the original algorithms. In a careful experimental evaluation, we compared our transformed algorithms with the original proposals, where the query evaluation is based on various concepts such as the X-tree, the R*-tree and the sequential scan. The traditional techniques are outperformed by factors of up to 33 for the X-tree and 54 for the R*-tree.

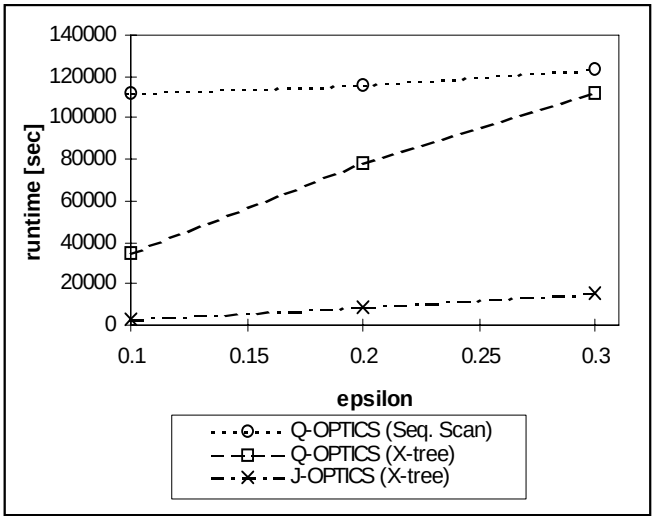

Figure 8. OPTICS for increasing query range

References

[1] Ankerst M., Breunig M. M., Kriegel H.-P., Sander J.: *'OPTICS: Ordering Points To Identify the Clustering Structure'*, Proc. ACM SIGMOD'99 Int. Conf. on Management of Data, Philadelphia, PA, 1999, pp. 49-60.

[2] Agrawal R., Gehrke J., Gunopulos D., Raghavan P.: *'Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications'*, Proc. ACM SIGMOD'98 Int. Conf. on Management of Data, Seattle, WA, 1998, pp. 94-105.

[3] Agrawal R., Imielinski T., Swami A.: *'Mining Association Rules between Sets of Items in Large Databases'*, Proc. ACM SIGMOD'93 Int. Conf. on Management of Data, Washington, D.C., 1993, pp. 207-216.

[4] Berchtold S., Keim D., Kriegel H.-P.: *'The X-Tree: An Index Structure for High-Dimensional Data'*, 22nd Int. Conf. on Very Large DataBases, 1996, Bombay, India, pp. 28-39.

[5] van den Bercken J., Seeger B., Widmayer P.:*'A General Approach to Bulk Loading Multidimensional Index Structures'*, 23rd Conf. on Very Large Databases, 1997, Athens, Greece.

[6] Berchtold S., Böhm C., Kriegel H.-P.: *'Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations'*, 6th. Int. Conf. on Extending Database Technology, 1998.

[7] Breunig S., Kriegel H.-P., Ng R., Sander J.: *'LOF: Identifying Density-Based Local Outliers'*, ACM SIGMOD Int. Conf. on Management of Data, Dallas, TX, 2000.

[8] Brinkhoff T., Kriegel H.-P., Seeger B.: '*Efficient Processing of Spatial Joins Using R-trees*', Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington D.C., 1993, pp. 237-246.

[9] Brinkhoff T., Kriegel H.-P., Seeger B.: '*Parallel Processing of Spatial Joins Using R-trees'*, Proc. 12th Int. Conf. on Data Engineering, New Orleans, LA, 1996.

[10] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.

[11] Ester M., Frommelt A., Kriegel H.-P., Sander J.: *'Algorithms for Characterization and Trend Detection in Spatial Databases'*, Proc. 4th Int. Conf. on Knowledge Discovery and Data Mining, New York, NY, 1998, pp. 44-50.

[12] Ester M., Kriegel H.-P., Sander J., Wimmer M. Xu X.: *'Incremental Clustering for Mining in a Data Warehousing Environment'*, Proc. 24th Int. Conf. on Very Large Databases, New York, NY, 1998, pp. 323-333.

[13] Ester M., Kriegel H.-P., Sander J., Xu X.: *'A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise'*, Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press, 1996, pp. 226-231.

[14] Faloutsos C., Lin K.-I.: *'FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Data'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose, CA, 1995, pp. 163-174.

[15] Gaede V., Günther O.:'*Multidimensional Access Methods'*, ACM Computing Surveys, Vol. 30, No. 2, 1998, pp.170-231.

[16] Guha S., Rastogi R., Shim K.: *'CURE: An Efficient Clustering Algorithms for Large Databases'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, WA, 1998, pp.73-84.

[17] Guttman A.: *'R-trees: A Dynamic Index Structure for Spatial Searching'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA, 1984, pp. 47-57.

[18] Huang Y.-W., Jing N., Rundensteiner E. A.:*'Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations'*, Proc. Int. Conf. on Very Large Databases, Athens, Greece, 1997, pp. 396-405.

[19] Hinneburg A., Keim D.A.: *'An Efficient Approach to Clustering in Large Multimedia Databases with Noise'*, Proc. 4th Int. Conf. on Knowledge Discovery & Data Mining, New York City, NY, 1998, pp. 58-65.

[20] Hattori K., Torii Y.: *'Effective algorithms for the nearest neighbor method in the clustering problem*'. Pattern Recognition, 1993, Vol. 26, No. 5, pp. 741-746.

[21] Huang, Z.: '*A Fast Clustering Algorithm to Cluster Very Large Categorical Data Sets in Data Mining*'. In Proc. SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, Tech. Report 97-07, UBC, Dept. of CS, 1997.

[22] Jagadish H. V.: *'A Retrieval Technique for Similar Shapes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Denver, CO, 1991, pp. 208-217.

[23] Jain A. K., Dubes R. C.: *'Algorithms for Clustering Data'*, Prentice-Hall, 1988.

[24] Keim D. A.: '*Visual Database Exploration Techniques'*, Proc. Tutorial Int. Conf. on Knowledge Discovery and Data Mining, Newport Beach, CA, 1997 (http://www.informatik.uni-halle.de/~keim/PS/KDD97.pdf).

[25] Koperski K., Han J.: *'Discovery of Spatial Association Rules in Geographic Information Databases'*, Proc. 4th Int. Symp. on Large Spatial Databases, Portland, ME, 1995, pp. 47-66.

[26] Knorr E.M., Ng R.T.: *'Finding Aggregate Proximity Relationships and Commonalities in Spatial Data Mining'*, IEEE Trans. on Knowledge and Data Engineering, Vol. 8, No. 6, 1996, pp. 884-897.

[27] Knorr E.M., Ng R.T.: *'Algorithms for Mining Distance-Based Outliers in Large Datasets'*, Proc. 24th Int. Conf. on Very Large DataBases, 1998, New York City, NY, pp. 392-403.

[28] Kaufman L., Rousseeuw P. J.: *'Finding Groups in Data: An Introduction to Cluster Analysis'*, John Wiley & Sons, 1990.

[29] Koudas N., Sevcik C.: '*Size Separation Spatial Join'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 324-335.

[30] Koudas N., Sevcik C.: '*High Dimensional Similarity Joins: Algorithms and Performance Evaluation'*, Proc. 14th Int. Conf on Data Engineering, Best Paper Award, Orlando, FL, 1998, pp. 466-475.

[31] Kriegel H.-P., Seidl T.: '*Approximation-Based Similarity Search for 3-D Surface Segments*', GeoInformatica Journal, Kluwer Academic Publishers, 1998, Vol.2, No. 2, pp. 113-147.

[32] Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z.: *'Fast Nearest Neighbor Search in Medical Image Databases'*, Proc. 22nd Int. Conf. on Very Large DataBases, Mumbai, India, 1996, pp. 215-226.

[33] Lin K., Jagadish H. V., Faloutsos C.: *'The TV-Tree: An Index Structure for High-Dimensional Data'*, VLDB Journal, 1995, Vol. 3, pp. 517-542.

[34] Lo M.-L., Ravishankar C. V.: '*Spatial Joins Using Seeded Trees'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Denver, 1994, pp. 517-542

[35] Lo M.-L., Ravishankar C. V.: '*Spatial Hash Joins'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1996, pp. 247-258.

[36] MacQueen, J.: *'Some Methods for Classification and Analysis of Multivariate Observations'*, 5th Berkeley Symp. Math. Statist. Prob., Vol. 1, pp. 281-297.

[37] Mitchell T.M.: '*Machine Learning'*, McCraw-Hill, 1997.

[38] Murtagh F.: *'A Survey of Recent Advances in Hierarchical Clustering Algorithms'*, The Computer Journal Vol. 26, No. 4, 1983, pp.354-359.

[39] Ng R. T., Han J.: *'Efficient and Effective Clustering Methods for Spatial Data Mining'*, Proc. 20th Int. Conf. on Very Large DataBases, Santiago de Chile, Chile, 1994, pp. 144-155.

[40] Patel J.M., DeWitt D.J., *'Partition Based Spatial-Merge Join'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1996, pp. 259-270.

[41] Piatetsky-Shapiro G., Frawley W. J.: *'Knowledge Discovery in Databases'*, AAAI/MIT Press, 1991.

[42] Richards A.J. *'Remote Sensing Digital Image Analysis. An Introduction'*, Berlin: Springer Verlag, 1983.

[43] Robinson J. T.: *'The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1981, pp. 10-18.

[44] Sellis T., Roussopoulos N., Faloutsos C.: '*The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*', Proc. 13th Int. Conf. on Very Large Databases, Brighton, 1987, pp.507-518.

[45] Sheikholeslami G., Chatterjee S., Zhang A.: *'WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases'*, Proc. Int. Conf. on Very Large DataBases, New York, NY, 1998, pp. 428 - 439.

[46] Sibson R.: *'SLINK: an optimally efficient algorithm for the single-link cluster method'*, The Computer Journal Vol. 16, No. 1, 1973, pp.30-34.

[47] Shim K., Srikant R., Agrawal R.: *'The ε-KDB tree: A Fast Index Structure for High-dimensional Similarity Joins'*, IEEE Int. Conf on Data Engineering, 1997, 301-311.

[48] Ullman J.D.: *'Database and Knowledge-Base System*', Vol. II,Compute Science Press, Rockville, MD, 1989.